Thread Synchronization in Java

Let's dive into thread synchronization in Java! When multiple threads access shared resources (like variables, collections, or files), you need to prevent race conditions—where threads interfere with each other, leading to unpredictable results. Java provides several tools for this: `synchronized`, `Lock`, and `Semaphore`. I'll explain each, show examples, and compare them so you can pick the right one for your needs.

---

1. `synchronized` Keyword

The `synchronized` keyword ensures only one thread can execute a block of code or method at a time. It's the simplest synchronization tool, built into Java's core.

How It Works

- Locks an object's monitor (every Java object has one).
- Other threads wait until the lock is released.

Example: Counter with `synchronized`

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
```

```
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount()); // Always 2000
    }
}
```

- Without `synchronized`, `count++` (read-modify-write) could overlap, giving a result less than 2000.
- `synchronized` locks the `Counter` instance, ensuring atomic updates.

Synchronized Block You can also lock specific code instead of a whole method:

```
public void increment() {
    synchronized (this) {
        count++;
    }
}
```

- Use a different object as the lock if you want finer control:

```
private final Object lock = new Object();
public void increment() {
    synchronized (lock) {
        count++;
    }
}
```

Pros and Cons

- **Pros**: Simple, built-in, good for basic use.
- **Cons**: No flexibility (e.g., can't interrupt a waiting thread), blocks until released.

---

2. `Lock` Interface (java.util.concurrent.locks)

The `Lock` interface (e.g., `ReentrantLock`) offers more control than `synchronized`. It's part of the `java.util.concurrent` package and was introduced to address `synchronized` limitations.

Key Features

- Explicit `lock()` and `unlock()` calls.
- Supports try-locks, timeouts, and interruptible locking.
- Fairness option (threads wait in order).

Example: Counter with `ReentrantLock`

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Counter {
    private int count = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock(); // Always unlock in finally
        }
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
```

```
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount()); // Always 2000
    }
}
```

- `try-finally` ensures the lock is released even if an exception occurs.

Advanced Features

- **Try Lock**: Non-blocking attempt to acquire the lock:

```
if (lock.tryLock()) {
    try {
        count++;
    } finally {
        lock.unlock();
    }
} else {
    System.out.println("Couldn't acquire lock");
}
```

- **Timeout**: Wait for a limited time:

```
if (lock.tryLock(1, TimeUnit.SECONDS)) { ... }
```

- **Interruptible**: Allow waiting threads to be interrupted:

```
lock.lockInterruptibly();
```

Pros and Cons

- **Pros**: Flexible, supports advanced features, explicit control.
- **Cons**: More verbose, manual unlock required (risk of forgetting).

3. `Semaphore`

A `Semaphore` controls access to a resource by maintaining a set of permits. It's great for limiting concurrency (e.g., max 5 threads can access a resource).

How It Works

- Threads acquire permits with `acquire()`.
- Release permits with `release()`.
- If no permits are available, threads wait.

Example: Limiting Database Connections

```
import java.util.concurrent.Semaphore;

class ConnectionPool {
    private final Semaphore semaphore = new Semaphore(3); // Max 3 connections

    public void connect() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName() + " connected");
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            semaphore.release();
            System.out.println(Thread.currentThread().getName() + " disconnected");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        ConnectionPool pool = new ConnectionPool();
        Runnable task = () -> pool.connect();

        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++) {
            threads[i] = new Thread(task, "Thread-" + i);
```

```
            threads[i].start();
        }
    }
}
```

- Only 3 threads can “connect”at once; others wait until a permit is released.

Advanced Features

- **Fairness**: `new Semaphore(3, true)` ensures FIFO order.
- **Try Acquire**: `semaphore.tryAcquire()` checks without blocking.

Pros and Cons

- **Pros**: Ideal for resource pools, flexible permit system.
- **Cons**: More complex than `synchronized` or `Lock`, not for simple mutual exclusion.

Combining with `ExecutorService`

Here’s how these tools work with a thread pool:

```
import java.util.concurrent.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SharedResource {
    private int value = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            value++;
        } finally {
            lock.unlock();
        }
    }

    public int getValue() {
```

```
        return value;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedResource resource = new SharedResource();
        ExecutorService executor = Executors.newFixedThreadPool(4);

        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                for (int j = 0; j < 100; j++) {
                    resource.increment();
                }
            });
        }

        executor.shutdown();
        executor.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Final value: " + resource.getValue()); // Always 1000
    }
}
```

Comparison

Feature	`synchronized`	`Lock` (ReentrantLock)	`Semaphore`
Ease of Use	High	Medium	Low
Flexibility	Low	High	Medium
Granularity	Method/block	Explicit	Permit-based
Interruptible	No	Yes	Yes
Fairness Option	No	Yes	Yes
Use Case	Simple mutual exclusion	Advanced control	Resource limiting

---

When to Use What?

- `synchronized`: Quick, simple mutual exclusion (e.g., protecting a single variable).

- `Lock`: Need try-locks, timeouts, or interruptibility (e.g., complex workflows).
- `Semaphore`: Limiting concurrent access to a resource (e.g., connection pools, rate limiting).